White Paper

Intel Software & Solutions Group

Greg Henry

Optimize for Intel® AVX Using Intel® Math Kernel Library's Basic Linear Algebra Subprograms (BLAS) with DGEMM Routine

This paper describes at a very high level some initial efforts at optimizing DGEMM and BLAS using the new Intel® Advanced Vector Extensions (Intel® AVX). The Intel® Math Kernel Library (Intel® MKL) has started optimizations with the Intel AVX and this paper talks about some of the preliminary ideas. Eventually we will target many domains of Intel MKL, as of today we have done initial implementations of BLAS, FFTs, and Vector Math libraries (VML). The initial implementations are fully functional. At the time of this writing, we have generated libraries for 64-bit operations in Windows*. Future Intel MKL support for Intel AVX will complete the feature set.

April 2008

Intel Corporation

Contents

Figures

Tables

Introduction

Although our initial efforts supports tunings and functionality in three areas, the Intel® Math Kernel Library (Intel® MKL), provides a broader set of functionality for scientific and engineering use. These are highlighted below:

- Linear Algebra – Basic Linear Algebra Subprograms (BLAS), LAPACK, ScaLAPACK, Sparse BLAS, Iterative Sparse Solvers, Preconditioners, Direct Sparse Solvers (PARDISO)
- FFTs – Both sequential and cluster FFTs
- Statistics – Vector Statics Library (VSL) and random number generators
- Vector Math – Vector Math Library (VML)
- PDEs – Poisson, Helmholtz solvers, trigonometric transforms
- Optimization – Trust Region Solvers

This paper starts with a brief outline of Intel MKL, then discusses generic performance related enhancements with regard to DGEMM, and then discusses a few of the new Intel® Advanced Vector Extensions (Intel® AVX) instructions, and wraps up in a summary. DGEMM is a double precision matrix-matrix multiplication algorithm, which is a key routine in the Basic Linear Algebra Subroutines (BLAS). More details of DGEMM shall be defined in the next section. However, as we shall show it is a fundamental routine in linear algebra and for this reason, we chose to start our optimizations here.

Many features of Intel MKL are designed to increase the ease of use of the library. For instance, for some routines, we provide different interfaces so that users can enjoy some flexibility. An example of this is the C interface to the BLAS or the FORTRAN 90 interface to LAPACK.

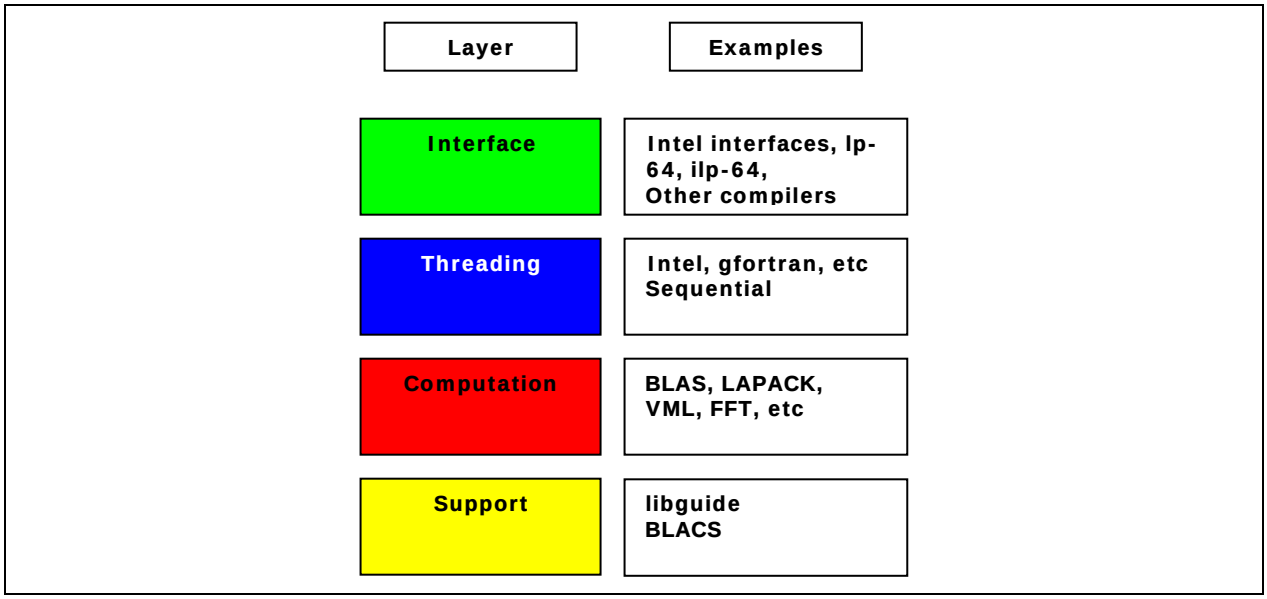
Another feature is that Intel MKL is designed to be compiler independent where possible, and components are split amongst independent library layers. There is an Intel® Compiler threading layer as well as a GNU compiler threading layer. Codes can be linked with either layer, and routines dependent on either the Intel compilers or GNU compilers are isolated into each layer. Figure 1 taken from [1] shows a little about the Intel MKL layer model. For more details, see the URL mentioned in [1].

Another feature is that a developer who links in Intel MKL doesn't need worry about tunings for a given processor; the Intel MKL library will do dispatching so that the best routines are used on any given processor at runtime. For instance, if an application using the BLAS is linked with a version of Intel MKL supporting Intel AVX on 64-bit Windows* running on a Intel® Xeon Processor 5400 series, and then the same binary is run on Intel processor with Intel AVX, then in each case it is possible that different code would be dispatched for the different processors. So, the same binary could be used in both runtime environments. Intel MKL makes it possible for a developer to always get the best performance possible, without requiring separate binaries for different processors.

Another feature is that Intel MKL provides competitive performance on non-Intel processors, thus making it easier for developers to use a single library for their products.

Intel MKL is also thread-safe and supports threading and multi-core optimization. By linking in Intel MKL, a developer can increase their performance in certain places where it makes sense to thread.

Figure 1: Layer Model for Intel® MKL



We now cover some specifics for DGEMM and the BLAS, and then later show how we can take advantage of the new Intel AVX in a DGEMM implementation.

DGEMM and the BLAS

BLAS stands for Basic Linear Algebra Subprograms [2], and it is one of first three functional components we have experimented with Intel AVX. Future experiments will be done with more routines in each component and as well as more Intel MKL components. Although to date we've experimented with three components (BLAS, FFTs and VML), this paper only discusses the BLAS. We started with the BLAS because the BLAS are the fundamental building blocks for many Linear Algebra routines.

The original idea behind the BLAS was to create a universal API so that developers using common Linear Algebra routines could take advantage of vendor tuned libraries. Intel MKL takes this vision one step further – not only can developers using the BLAS take advantage of new Intel extensions like Intel AVX, but they can create binaries that automatically check the CPU and take advantage of whichever processor specific optimizations are possible. Therefore, binaries created with Intel MKL would dispatch fast Intel AVX code on future processors that support Intel AVX, and dispatch fast Intel® SSE2 code in other places. As such, binaries built with our preliminary code will run fine on processors, which exist today, as well as run, via simulation, on future extensions such as Intel AVX. This takes the entire issue of processor-specific optimization and simplifies it to a point where the developer never has to worry about it.

The BLAS are further divided into "levels" 1 through 3. If N represents a vector size, or the order of a matrix, then Table 1 summarizes the characteristics of the BLAS. Note that the Level number is defined by the order of magnitude on the number of floating-point operations, but that the ratio of data movement to floating-point operations is most favorable in the Level 3 BLAS.

Table 1. Levels of BLAS

Level	Data Movement	Floating-Point Operations	Example
Level 1 BLAS	O(N)	O(N)	DDOT
Level 2 BLAS	O(N²)	O(N²)	DGEMV
Level 3 BLAS	O(N²)	O(N³)	DGEMM

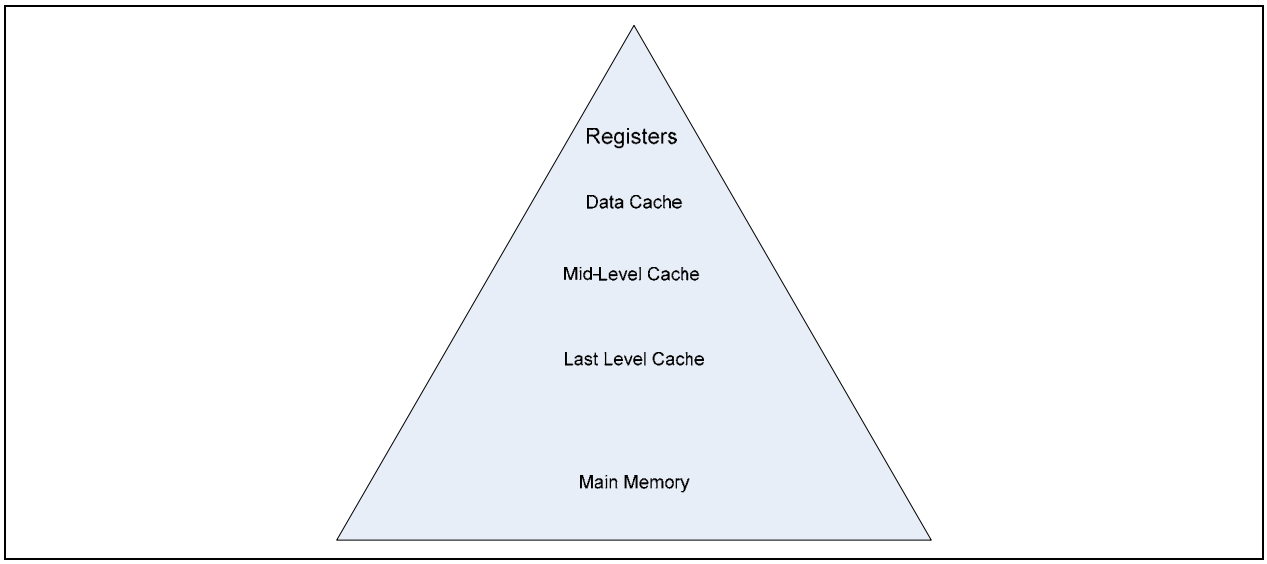
There is a growing gap between memory performance and the computational speed of a modern processor. This is largely because faster memory is more expensive and harder to implement in large quantities. This has been the case for years, and the solution is to build multi-core machines with a memory hierarchy of multiple caches plus main memory. Given this hierarchy, algorithms can incorporate data blocking to take advantage of the different levels of the hierarchy.

We view the memory hierarchy as a pyramid [3]. The top of the pyramid contains the memory that is the fastest, and also the smallest, and the bottom of the pyramid contains the memory that is the slowest but also the largest. The best routines to take advantage of such a pyramid are the Level-3 BLAS. The reason they can take the greatest advantage of this hierarchy is that they have an order of magnitude more computation than memory interaction, their memory interaction can be organized in a fashion such that it takes advantage of the various stages of the pyramid.

The pyramid concept does not completely map to modern multi-core optimization, however. The difference is that on multi-core, there are often many cores at the top of the pyramid, and they may have a shared cache between them. The bottom of the pyramid may also contain a NUMA-based model. NUMA stands for Non-Uniform Memory Access.

We consider the highest level to be machine registers. Register data movement keeps pace with processor clock rates. The next level is the data cache unit (DCU), then the mid-level cache, then the last level cache, and finally the main memory itself. The access to the memory, however, is based on some proximity – some memory is faster to access from some processors than others (in the case of NUMA) – although all of memory can be accessed by all processors.

Figure 2: Memory Hierarchy Pyramid



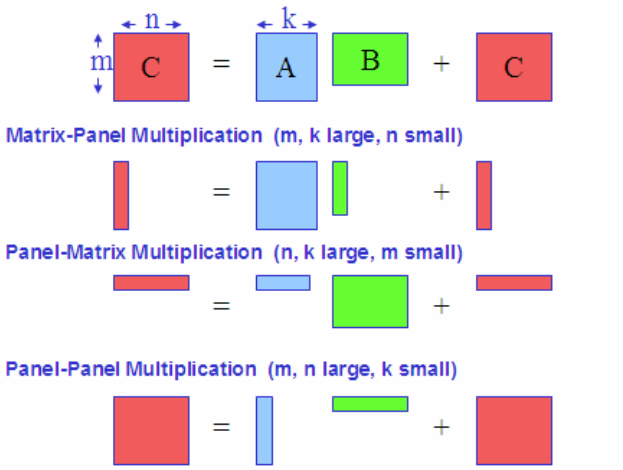
The higher up on the pyramid, the more valued the resource. In an ideal world, you'd have many more flops than memory movements, because memory movements can be slow, especially if they are misses in a given level of the hierarchy. Level 3 BLAS routines have O(N^3) flops and O(N^2) data movement, so it is possible to block for all levels of the memory hierarchy.

As mentioned, multi-core and NUMA does change Figure 2 slightly. In particular, each core has its own data cache and mid-level cache, but the last level cache is shared amongst the cores. The last level cache may be inclusive of the mid-level cache, but the mid-level cache may not be inclusive of the data cache.

In [4], it is shown that the Level-3 BLAS can be constructed in terms of matrix-matrix multiply. This is not enough for full performance, but it does show that matrix-matrix multiplication is the most important of the Level 3 BLAS. In particular, this paper focuses on Double precision GEneral Matrix Matrix operation (DGEMM). DGEMM takes three matrices (A, B, C) and updates C with beta*C + alpha*op(A)*op(B) where alpha and beta are scalars and op(*) means that one can insert the transpose, conjugate transpose, or just the matrix as is. Since blocking or moving the matrices from one level of cache to another often involves copying, without loss of generality, we can ignore the op(*) and just focus on C = C + A*B, based on the assumption that if we can do this operation fast, we can incorporate a transpose or a scalar update equally as fast.

In [6], DGEMM can be broken down into three different types of matrix manipulations.

Figure 3: Different Flavors of DGEMM



One sometimes views the memory hierarchy as a series of steps. At any given time, some data may be on one step and it must move up or down the hierarchy to another step. The crux behind [6] is the observation that at any given step of the memory hierarchy, one can ignore the other steps and just focus on making the blocking choices as wisely as possible so as to minimize data movement overhead required to get the next step. This results in a family of algorithms, where one can choose the fastest method for solving the problem overall.

In Intel MKL, we use a similar approach. Note that this sort of approach is also taken in some recent successful works such as [7].

DGEMM and Intel® AVX

Intel AVX essentially doubles the register width of the Intel SSE2 registers. DGEMM kernels within Intel MKL are mostly made up of three instructions:

vmovaps
vaddp
vmulpd

These are natural extensions to the Intel SSE2 instructions:

movaps
addpd
mulpd

That is, instead of fitting two double precision elements into the 128-bit Intel SSE2 registers, we are now able to fit 4 double precision elements into the 256-bit Intel AVX registers. Note that for Intel SSE2 manipulations on modern Intel® Core™2 hardware, the processors can execute (on different ports) both addpd and mulpd at the same time. This means that on Intel Core 2 hardware, the peak floating-point rate is four floating-point operations per cycle (two double precision adds from addpd, and two double precision multiplies from mulpd). The number of single precision operations would be twice this amount, but we will focus on double precision in this paper.

Intel AVX doubles the theoretical floating-point bandwidth. This is because the hardware implementation for Intel AVX will also be able to execute a vaddpd and a vmulpd in the same cycle. However, since the register size is twice as big, this means that the theoretical floating-point rate doubles. Of course, the register movement rate from the vmovaps is also doubled.

Another instruction that is useful is "vbroadcastsd" which enables us to load the same value into all four 64-bit locations of the Intel AVX registers simultaneously.

Intel AVX allows for a 3rd destination parameter to be specified. This brings much more flexibility than Intel SSE2 provided. So, one can execute a "vmulpd ymm0, ymm1, ymm2" and reuse both input parameters to the multiplication again in a different instruction (the destination register is usually the first register, ymm0, on Windows*, and the last register, ymm2, on Linux* – since our first experiments have been with Windows*, any instructions in this paper will use the Windows* syntax). Intel SSE2 would require the addition of register movement instructions to store off registers that needed to be preserved in subsequent steps.

In performance simulations, we are able to see quite close to the expected doubling of performance with Intel AVX over similar Intel SSE2 code.

We can load approximately 32 bytes per cycle (256 bits, one extended register) if the data is cache resident and there are no other issues (bank conflicts or TLB or cache misses). Just as one can block for the caches, one can block for the registers (register blocking). We can assume register blocking because we can process 512 bits of data per cycle with a simultaneous vaddpd and vmulpd. This means that we can compute faster than we can load the data. Therefore, we'd like to get re-use out of register blocking, so that loads from A or B are reused as much as possible during temporary calculations of C.

To illustrate register blocking in practice, let us first consider an example DGEMM inner kernel where no blocking is done. As mentioned, the new Intel AVX registers can hold four 64-bit floating-point values. If we have one register that loads from A, one register that loads from B, and one register that accumulates 4 dot-products simultaneously, the code might look something like:

```
vmovapd ymm0, [rax]
vbroadcastsd ymm1, [rbx]
vmulpd ymm0, ymm0, ymm1
vaddpd ymm2, ymm2, vmm0
```

In this case, let us assume rax contains the address of A, and rbx contains the address of B. Notice that we're only getting 1 B value at a time, and multiplying it by 4 different A values. We are using one register for A values (ymm0), one register for B values (ymm1), and one register for C values (ymm2). If rax and rbx are incremented appropriately, and a loop count is created, at the end of loop, one will have completed four rows of A*B. This code is not register blocked, and the problem with this code is that there are 2 loads per pair of adds and multiplies. The adds and multiplies can execute together in a single cycle, but the resulting loop will be load bound because the loads of A and B will require more than a single cycle.

Let us extend this strategy to the general case now involving register blocking. Each load of A requires one register in this strategy. Each load of B (even though we are using broadcast) requires one register. So suppose α is the number of registers we put aside for A loads. In addition, suppose β is the number of registers put aside for B loads. Notice that this strategy requires at least αβ registers for storing the dot products for C as well. The total register use is then α + β + αβ. The number of multiplies, or the number of adds, however is going to αβ. Our objective is to have as many more adds and multiplies as we can compared to loads, subject to only having 16 registers total. The number of loads is α + β. Some possible solutions to this problem are (α=4,β=2), (α=2,β=4), or (α=3,β=3). In any of these cases, by re-using the registers that are put aside for A and B loads, we can organize our inner loops so that unlike the first example given, we have more multiplies and adds than we have loads.

This form of register blocking can be viewed simply as unrolling the innermost loop. Once the unroll factors are determined, the only thing left to decide is the order of computation. Intel MKL will block for the registers, the caches, and the TLB, but there are also other considerations. For instance, the DCU has eight memory banks, so we try to avoid accessing the same banks close together. Another thing we can do is prefetch values that we will need later. There are many different forms of prefetch instructions. The simplest form is to load values in a register in advance of when they are required. Of course, the processor will do out of order execution anyhow, but since this requires keeping track of many resources, and the inner most loop, once unrolled fully, may be quite long, and in many cases it helps to do this.

In the register blocking case, we tried to minimize the number of loads and stores compared to the number of multiplies and adds. In the full DGEMM case, C = C + A*B, where C is MxN and A is MxK and B is KxN, it is important to achieve high performance when both M and N are much bigger than K, as this case frequently comes up in applications. We tested our new DGEMM, under simulation with preliminary simulators, when M=N=1024 and K=128. Theoretically, we know that this should run with half the clocks as any existing hardware today. According to simulation, we see a 1.9x improvement for this specific case. Of course, the simulators are under development.

Summary

We have started optimizing Intel MKL in three functional areas using Intel AVX. Our experiences so far are quite positive. We can see direct evidence of the doubling of the theoretical floating-point capacity with the new instructions, as well as the convenience of being able to use these natural extensions of Intel SSE2. E.g., we have had much success so far with the most important BLAS function: DGEMM. For this kernel, we are seeing close to twice the performance over Intel SSE2 code for matrices fitting in the

data cache unit. Over time, Intel MKL will provide more Intel AVX optimizations, and they will all automatically dispatch on processors with Intel AVX instruction capacity and for the other processors, they will still dispatch the existing high quality Intel MKL code.

References

[1] Intel Performance Libraries: Multicore-ready Software for Numeric Intensive Computation, by I. Burylov, M. Chuvelev, B. Greer, G. Henry, S. Kuznetsov, B. Sabanin, Intel Technical Journal, Q4 Vol. 11 Issue 4, November 2007, http://www.intel.com/technology/itj/2007/v11i4/4-libraries/1-abstract.htm

[2] A Set of Level 3 Basic Linear Algebra Subprograms, by J.J. Dongarra, J. Du Croz, I.S. Duff, and S. Hammarling, TR ANL-MCS-TM-88, Argonne National Laboratory, Argonne, IL, 1988.

[3] Matrix Computations, by G.H. Golub and C. Van Loan, The Johns Hopkins University Press, Baltimore MD, 2nd Ed., 1989.

[4] GEMM-based Level 3 BLAS, by B. Kågström and C. Van Loan, TR CTC91TR47, Cornell Theory Center, Ithaca NY 1991.

[5] FLAME: Formal Linear Algebra Methods Environment, by John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn, ACM Transactions on Mathematical Software, Volume 27, No. 4, Pages 422-455, Dec. 2001.

[6] High-Performance Matrix Multiplication Algorithms for Architectures with Hierarchical Memories, by John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn, FLAME Working Note #4, June 2001.

[7] Anatomy of High Performance Matrix Multiplication, by Kazushige Goto and Robert van de Geijn, ACM Transactions on Mathematical Software, Vol 34, No 3, 2008.

About the Author

Greg Henry is a Principal Engineer in the Intel MKL team in the Developer Products Division of the Software Solutions Group at Intel Corporation. His research interests are linear algebra, parallel computing, numerical analysis, scientific computing, and all things relevant to Intel MKL. He received his Ph.D. degree from Cornell University in Applied Mathematics and started working at Intel in August 1993. Greg has three children and a wonderful wife and writes novels as a hobby. His e-mail is greg.henry at intel.com.